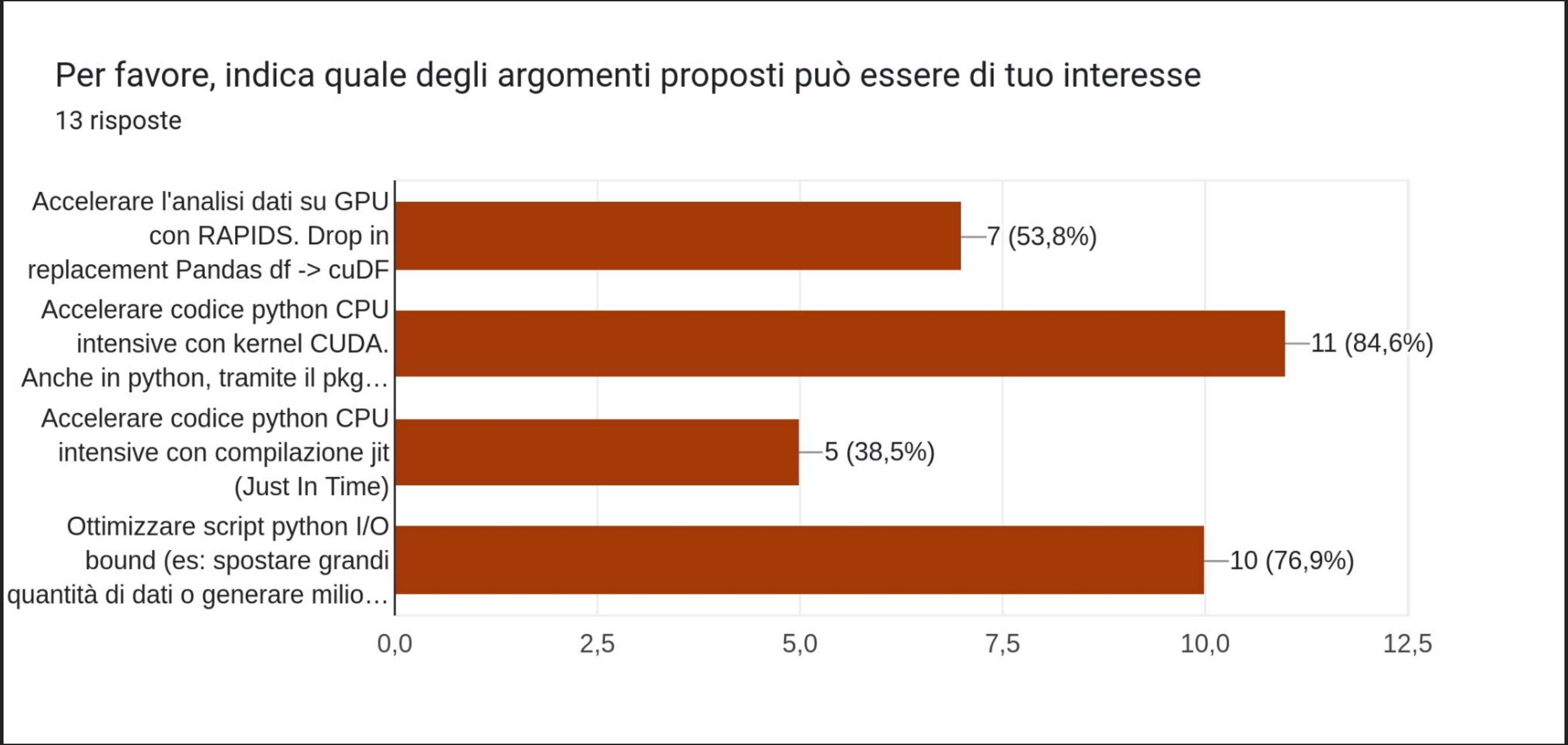
# CUDA Python

```
git clone https://github.com/fvisconti/cupy_course.git
```

/me: francesco.visconti@inaf.it

# Survey results



## Python and CUDA: a tough story

- NVIDIA proposes the use of CUDA from Python through the **`numba`** package
- `Numba` was initially born as an open-source **CPU-only** accelerator, while the modules that interfaced with the GPU (`accelerate`) were proprietary. Then, they decided to **merge into a single open-source project**
- **`accelerate`** gets rewritten and becomes `PyCULib`, which integrates, for example, `cuBLAS` and other GPU-accelerated algebra libraries.
- `PyCULib` <u>has not been officially updated</u> for 6 years
- <u>**`cupy`**</u> to the rescue!

## Warm up: performance

Let's go and see what happens with this code

```python
import cupy as cp
import numpy as np
import time
# NumPy and CPU Runtime
cpus = time.perf_counter()
x_cpu = np.ones((1000, 1000, 200))
cpue = time.perf_counter()
print(f"Time consumed by numpy: {cpue - cpus}")
# CuPy and GPU Runtime
s = time.perf_counter()
x_gpu = cp.ones((1000, 1000, 200))
e = time.perf_counter()
print(f"\nTime consumed by cupy: {e - s}")

print(f"\nspeed-up is by a factor {(cpue-cpus)/(e-s)}")
```

# Warm up: performance

Let's go and see what happens with this code.

It may take several seconds when calling a CuPy function for the first time in a process. This is because the CUDA driver creates a CUDA context during the first CUDA API call in CUDA applications.

CuPy uses on-the-fly kernel synthesis: when a kernel call is required, it compiles a kernel code optimized for the shapes and dtypes of given arguments, sends it to the GPU device, and executes the kernel

```python
import cupy as cp
import numpy as np
import time
# NumPy and CPU Runtime
cpus = time.perf_counter()
x_cpu = np.ones((1000, 1000, 200))
cpue = time.perf_counter()
print(f"Time consumed by numpy: {cpue - cpus}")
# CuPy and GPU Runtime
s = time.perf_counter()
x_gpu = cp.ones((1000, 1000, 200))
e = time.perf_counter()
print(f"\nTime consumed by cupy: {e - s}")

print(f"\nspeed-up is by a factor {(cpue-cpus)/(e-s)}")
```

# Warm up: measure correctly

*Because GPU executions run asynchronously with respect to CPU executions, a common pitfall in GPU programming is to mistakenly measure the elapsed time using CPU timing utilities (such as* **`time.perf_counter()`** *from the Python Standard Library or the `%timeit` magic from `IPython`), which have no knowledge in the GPU runtime.*

**Use the internal benchmark function instead!**

```python
import cupy as cp
import numpy as np
import time
# NumPy and CPU Runtime
cpus = time.perf_counter()
x_cpu = np.ones((1000, 1000, 200))
cpue = time.perf_counter()
print(f"Time consumed by numpy: {cpue - cpus}")
# CuPy and GPU Runtime
s = time.perf_counter()
x_gpu = cp.ones((1000, 1000, 200))
e = time.perf_counter()
print(f"\nTime consumed by cupy: {e - s}")

print(f"\nspeed-up is by a factor {(cpue-cpus)/(e-s)}")
```

```python
from cupyx.profiler import benchmark
import cupy as cp
import numpy as np

def cpu_init():
      return np.ones((1000, 1000, 200))

def gpu_init():
      return cp.ones((1000, 1000, 200))

cpu_bench = benchmark(cpu_init, n_repeat=20)
gpu_bench = benchmark(gpu_init, n_repeat=20)
```

# Warm up: measure correctly

What `cupyx.profiler.benchmark` internally do:

```python
import time
start_gpu = cp.cuda.Event()
end_gpu = cp.cuda.Event()

start_gpu.record()
start_cpu = time.perf_counter()
out = my_func(a)
end_cpu = time.perf_counter()
end_gpu.record()
end_gpu.synchronize()
t_gpu = cp.cuda.get_elapsed_time(start_gpu,
end_gpu)
t_cpu = end_cpu - start_cpu
```

```python
from cupyx.profiler import benchmark
import cupy as cp
import numpy as np

def cpu_init():
      return np.ones((1000, 1000, 200))

def gpu_init():
      return cp.ones((1000, 1000, 200))

cpu_bench = benchmark(cpu_init, n_repeat=20)
gpu_bench = benchmark(gpu_init, n_repeat=20)
```

# Cupy basics

1. `cupy` can be used in many cases as a replacement for `numpy`!
2. *current device* concept
   - All CuPy operations (except for multi-GPU features and device-to-device copy) are performed on the currently active device.
   - device can be allocated

```
with cp.cuda.Device(1):
     x = cp.array([1, 2, 3, 4, 5])
x.device
<CUDA Device 1>
```

## Cupy basics

Data transfer: move data between host and devices.

```
x_cpu = np.array([1, 2, 3])
x_gpu = cp.asarray(x_cpu) # move the data to the current device.
```

```
with cp.cuda.Device(0):
     x_gpu_0 = cp.ndarray([1, 2, 3]) # create an array in GPU 0
with cp.cuda.Device(1):
     x_gpu_1 = cp.asarray(x_gpu_0) # move the array to GPU 1
```

```
x_cpu = x_gpu.get() # move the array to the host
x_cpu = x_gpu.asnumpy() # same
```

## Cupy basics

Agnostic code: exploit compatibility with numpy

```python
import cupy as cp
import numpy as np
from cupyx.profiler import benchmark

# Stable implementation of log(1 + exp(x))
def softplus(x):
      xp = cp.get_array_module(x)
      print("Using:", xp.__name__)
      return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))

x = np.random.random(int(1e5))
x_gpu = cp.asarray(x)

cpu_bench = benchmark(softplus, (x,), n_repeat=10)
gpu_bench = benchmark(softplus, (x_gpu,), n_repeat=10)
```

## User defined kernels

CuPy provides easy ways to define three types of CUDA kernels:

- elementwise kernels → `cp.ElementwiseKernel()` class
- reduction kernels → `cp.ReductionKernel()` class
- raw kernels → `cp.RawKernel()` class

## User defined kernels

- An elementwise kernel refers to a function or operation that is applied independently to each element of one or more input arrays. The same operation is performed simultaneously on multiple data elements in parallel.
- A reduction kernel refers to a function or operation that combines multiple elements of an input array into a single result by applying a reduction operation such as the sum, minimum, maximum, or average of the elements. Reduction kernels are used to efficiently compute global aggregates or statistics from large arrays.
- A raw kernel is a set of operation on data entirely defined by the user.

Ref: https://docs.cupy.dev/en/stable/user_guide/kernel.html

## User defined kernels

Instead of using the mentioned `cupy` classes, and in order to keep things more *pythonic*, we're going to make use of two handy tools from cupy, both usable as classical python ***decorators***:

- kernel fusion
- JIT kernel definition.

## Kernel fusion

Inside the function, we perform elementwise computations on the input arrays, which include sin(), cos(), and sqrt() operations. The @cp.fuse() decorator allows these computations to be efficiently combined into a single kernel, reducing memory transfers and kernel launch overhead.

@cp.fuse() decorator is particularly useful when there are multiple elementwise computations to be performed, as it helps optimize the execution and reduce overhead.

```python
import cupy as cp
from cupyx.profiler import benchmark

# Define input arrays
a = cp.arange(10)
b = cp.arange(10, 20)
c = cp.arange(20, 30)

# Define an elementwise computation using @cp.fuse() decorator
@cp.fuse()
def elementwise_computation(x, y, z):
      return cp.sin(x) + cp.cos(y) / cp.sqrt(z)

# Invoke the elementwise computation
bench = benchmark(elementwise_computation, (a, b, c),
n_repeat=10)
```

## Kernel fusion

another example not involving analytical math functions

```python
import cupy as cp
from cupyx.profiler import benchmark

# Define input arrays
b = cp.arange(10, 20)
c = cp.arange(20, 30)

@cp.fuse(kernel_name='squared_diff')
def squared_diff(x, y):
      return (x - y) * (x - y)

print(benchmark(squared_diff, (b, c), n_repeat=10))
```

## JIT kernel definition

Both styles to launch the kernel, as shown above, are supported. The first two entries are the grid and block sizes, respectively. grid ( RawKernel style (128,) or Numba style [128]) is the sizes of the grid, i.e., the numbers of blocks in each dimension; block ((1024,) or [1024]) is the dimensions of each thread block.

The compilation will be deferred until the first function call. CuPy's JIT compiler infers the types of arguments at the call time, and will cache the compiled kernels for speeding up any subsequent calls.

```
import cupy as cp
from cupyx import jit

@jit.rawkernel()
def elementwise_square(x, y, size):
      tid = jit.blockIdx.x * jit.blockDim.x + jit.threadIdx.x
      ntid = jit.gridDim.x * jit.blockDim.x
for i in range(tid, size, ntid):
      y[i] = x[i] * x[i]

size = cp.uint32(2 ** 22)
x = cp.arange(size, dtype=cp.float32)
y = cp.empty((size,), dtype=cp.float32)

elementwise_square((128,), (1024,), (x, y, size)) # RawKernel
style
assert (y == x * x).all()

elementwise_square[128, 1024](x, y, size) # Numba style
assert (y == x * x).all()
```

## JIT kernel definition

Typing rule

The types of local variables are inferred at the first assignment in the function. The first assignment must be done at the top-level of the function; in other words, it must *not* be in `if/else` bodies or `for`-loops.

Limitations

JIT does not work inside Python’s interactive interpreter as the compiler needs to get the source code of the target function.

## Memory Management

CuPy uses *memory pool* for memory allocations by default. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

There are two different memory pools in CuPy:

- **Device memory pool** (GPU device memory), which is used for GPU memory allocations.
- **Pinned memory pool** (non-swappable CPU memory), which is used during CPU-to-GPU data transfer.

# Memory Management

The a_cpu array resides on CPU, so the mempool does not know him

```
import cupy
import numpy

mempool = cupy.get_default_memory_pool()
pinned_mempool = cupy.get_default_pinned_memory_pool()

# Create an array on CPU.
# NumPy allocates 400 bytes in CPU (not managed by CuPy memory
pool).
a_cpu = numpy.ndarray(100, dtype=numpy.float32)
print(a_cpu.nbytes) # 400

# You can access statistics of these memory pools.
print(mempool.used_bytes()) # 0
print(mempool.total_bytes()) # 0
print(pinned_mempool.n_free_blocks()) # 0
```

## Memory Management

Now the array a is brought to GPU

```
import cupy
import numpy

mempool = cupy.get_default_memory_pool()
pinned_mempool = cupy.get_default_pinned_memory_pool()

a = cupy.array(a_cpu)
print(a.nbytes) # 400
print(mempool.used_bytes()) # 512
print(mempool.total_bytes()) # 512
print(pinned_mempool.n_free_blocks()) # 1
```

## Memory Management

When the array goes out of scope, the allocated device memory is released and kept in the pool for future reuse.

You can clear the memory pool by calling `free_all_blocks`.

```python
a = None # (or `del a`)
print(mempool.used_bytes()) # 0
print(mempool.total_bytes()) # 512
print(pinned_mempool.n_free_blocks()) # 1

mempool.free_all_blocks()
pinned_mempool.free_all_blocks()
print(mempool.used_bytes()) # 0
print(mempool.total_bytes()) # 0
print(pinned_mempool.n_free_blocks()) # 0
```

## Memory Management: set memory limit for your code

Memory limits can be set via environment variables

```
$ export CUPY_GPU_MEMORY_LIMIT="1073741824"
$ export CUPY_GPU_MEMORY_LIMIT="50%"
```

## Memory Management: set memory limit for your code

Or calling the proper APIs (which overrides env vars). This is also nice since it lets you set a limit for each device.

```python
import cupy

mempool = cupy.get_default_memory_pool()

with cupy.cuda.Device(0):
      mempool.set_limit(size=1024**3) # 1 GiB

with cupy.cuda.Device(1):
      mempool.set_limit(size=2*1024**3) # 2 GiB
```

## Best practices

As you probably know, it's important to identify **hotspot** in your code, before even starting to define an optimization strategy.
The code needs to be ***profiled*** in order to know what code section to attack first: profiling tools help you to see how much time do you spend event in a single line of code, and `cupy` has a wonderful practical function to help you do that.

## Best practices

We saw in first slides about the `cupyx.profiler.benchmark` tool. Let's recall what it does internally.

This tool runs a few warm-up runs to reduce timing fluctuation and exclude the overhead in first invocations.

```
import cupy as cp
import time
start_gpu = cp.cuda.Event()
end_gpu = cp.cuda.Event()

start_gpu.record()
start_cpu = time.perf_counter()
out = my_func(a)
end_cpu = time.perf_counter()
end_gpu.record()
end_gpu.synchronize()
t_gpu = cp.cuda.get_elapsed_time(start_gpu, end_gpu)
t_cpu = end_cpu - start_cpu
```

## One-time Overhead

### Context initialization

It may take several seconds when calling a `CuPy` function for the first time in a process. This is because the CUDA driver **creates a CUDA context** during the first CUDA API call in CUDA applications.

### Kernel compilation

**When a kernel call is required**, it compiles a kernel code optimized for the dimensions and `dtypes` of the given arguments, sends them to the GPU device, and executes the kernel.

CuPy **caches the kernel code** sent to GPU device within the process, which reduces the kernel compilation time on further calls.

The compiled code is also cached in the directory `${HOME}/.cupy/kernel_cache` (the path can be overwritten by setting the **`CUPY_CACHE_DIR`** environment variable). **This allows reusing the compiled kernel binary across the process**.

# Differences with numpy

**Data types in cupy arrays cannot be non numeric (strings, objects) in cupy!**

Although you can generally use cupy as a replacement for numpy, there are some differences you may encounter.

Let's see some of them:

- casting from float to integer.

```
np.array([-1], dtype=np.float32).astype(np.uint32)
→ array([4294967295], dtype=uint32)
cupy.array([-1], dtype=np.float32).astype(np.uint32)
→ array([0], dtype=uint32)

np.array([float('inf')], dtype=np.float32).astype(np.int32)
→ array([-2147483648], dtype=int32)
cupy.array([float('inf')], dtype=np.float32).astype(np.int32)
→ array([2147483647], dtype=int32)
```

## Differences with numpy

**Data types in cupy arrays cannot be non numeric (strings, objects) in cupy!**

### Reductions output

numpy returns a scalar when performing reductions, while cupy returns a zero dimensional array.

Since cupy's scalars are aliases for numpy's, returning scalars would require synchronization. You can always cast yourself to a scalar if needed.

```python
print(type(np.sum(np.arange(3))))
print(type(cp.sum(cp.arange(3))))
```

```
<class 'numpy.int64'>
<class 'cupy.ndarray'>
```

## Concurrently executes two kernels

Suppose you have these two different kernels ⟶

If they were totally unrelated, there would be no need to run the first and wait for its completion: let's use `Streams` then.

```
@cp.fuse()
def elementwise_computation(x, y, z):
      return cp.sin(x) + cp.cos(y) / cp.sqrt(z)

@cp.fuse(kernel_name='squared_diff')
def squared_diff(x, y):
      return (x - y) * (x - y)
```

## Concurrently executes two kernels

Using the with statement we implicitly execute the CUDA operations in the code block using that stream. The result of doing this is that the second kernel, i.e. `squared_diff`, does not need to wait for `elementwise_computation` to finish before being executed.

```
with cp.cuda.Stream() as stream1:
      elementwise_computation(...)
with cp.cuda.Stream() as stream2:
      squared_diff(...)
```

## Concurrently executes two kernels

What if we instead have to wait for stream1 to finish? We have the `synchronize()` primitive to help.

This way though, we have to wait for the whole operations in `stream1` to finish. What if we just want to wait for a particular event to happen and then go on? Yes, `Events`! You already faced them in the **Best Practices** section, they are used in the `benchmark` code.

```
stream1.synchronize()
```

```
stream1 = cupy.cuda.Stream()
stream2 = cupy.cuda.Stream()
sync_point = cupy.cuda.Event()

with stream1:
    elementwise_comp(...)
    sync_point.record(stream=stream1)
    elementwise_comp(...)
with stream2:
    stream2.wait_event(sync_point)
    squared_diff(...)
```

```
git clone https://github.com/fvisconti/cupy_course.git
```

/me: francesco.visconti@inaf.it